ΤΕΧΝΙΚΕΣ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΟΥΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Κληρονομικότητα

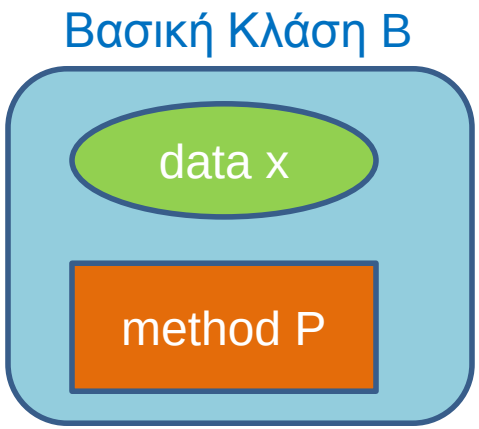
Πολυμορφισμός – Late Binding

Κληρονομικότητα

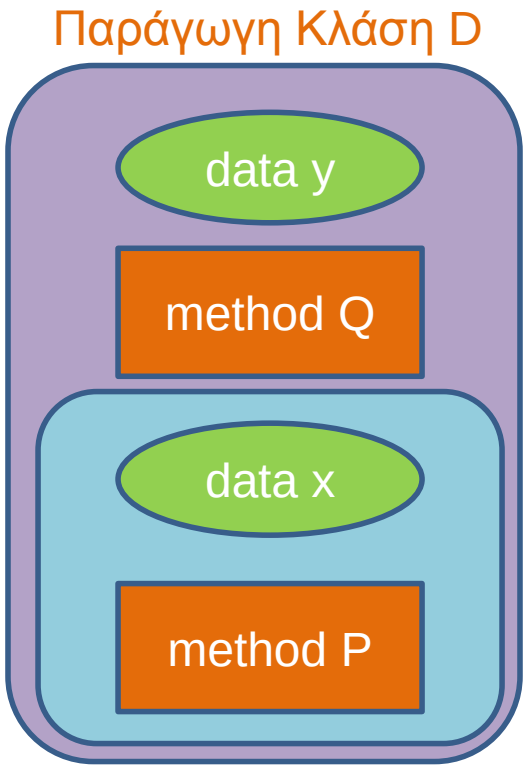
- Η κληρονομικότητα είναι κεντρική έννοια στον αντικειμενοστραφή προγραμματισμό.
- Η ιδέα είναι να ορίσουμε μια γενική κλάση που έχει κάποια χαρακτηριστικά (πεδία και μεθόδους) που θέλουμε και μετά να ορίσουμε εξειδικευμένες παραλλαγές της κλάσης αυτής στις οποίες προσθέτουμε ειδικότερα χαρακτηριστικά.
  - Οι εξειδικευμένες κλάσεις λέμε ότι κληρονομούν τα χαρακτηριστικά της γενικής κλάσης

Κληρονομικότητα

Έχουμε μια Βασική Κλάση (Base Class) Β, με κάποια πεδία και μεθόδους.



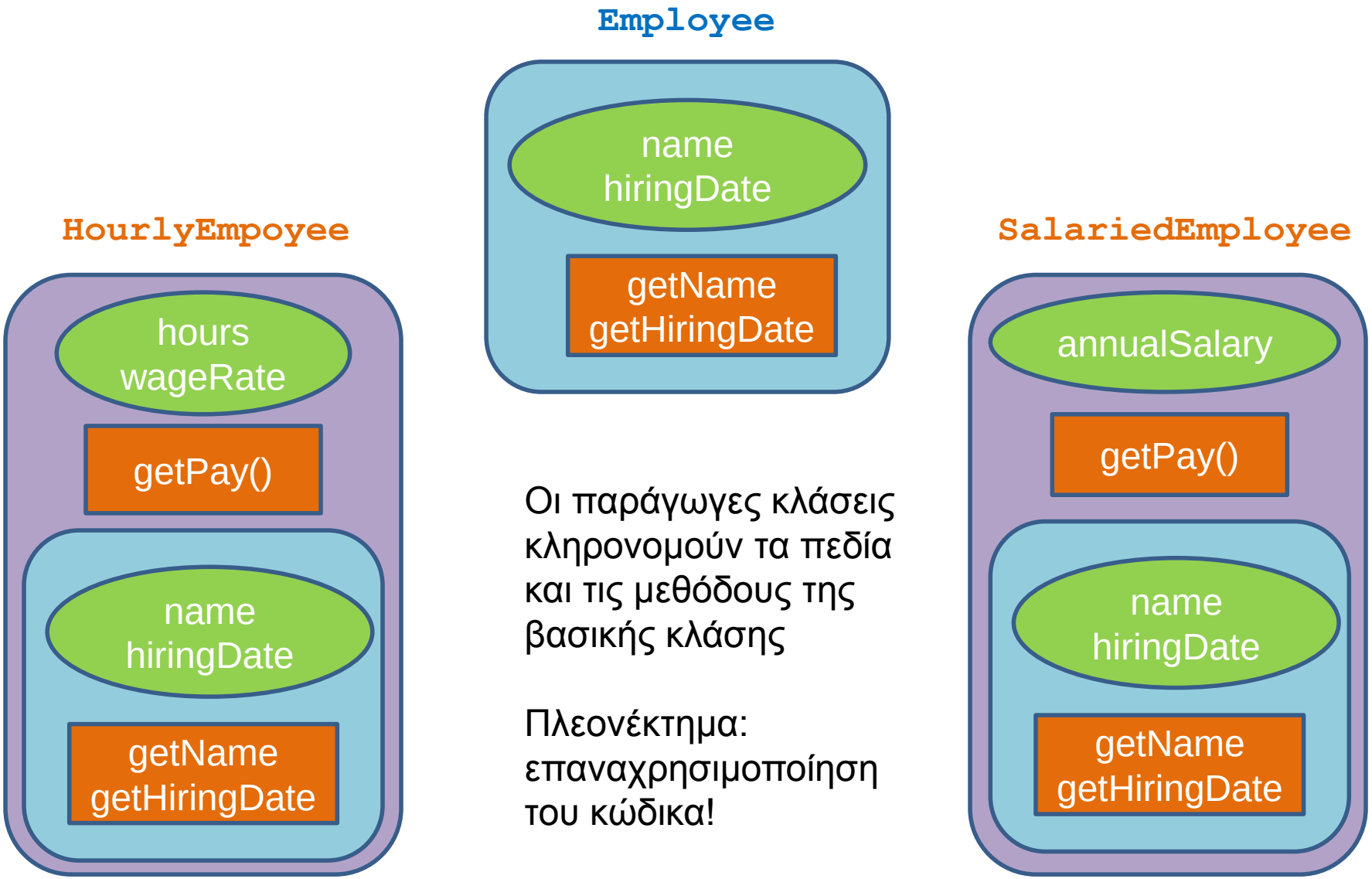
Θέλουμε να δημιουργήσουμε μια νέα κλάση D η οποία να έχει όλα τα χαρακτηριστικά της Β, αλλά και κάποια επιπλέον.



Αντί να ξαναγράψουμε τον ίδιο κώδικα δημιουργούμε μια Παράγωγη Κλάση (Derived Class) D, η οποία κληρονομεί όλη τη λειτουργικότητα της Βασικής Κλάσης Β και στην οποία προσθέτουμε τα νέα πεδία και μεθόδους.

Αυτή διαδικασία λέγεται κληρονομικότητα

Παράδειγμα



Οι παράγωγες κλάσεις κληρονομούν τα πεδία και τις μεθόδους της βασικής κλάσης

Πλεονέκτημα: επαναχρησιμοποίηση του κώδικα!

Ιεραρχία κλάσεων

- Μέσω της σχέσεως κληρονομικότητας μπορούμε να ορίσουμε μια ιεραρχία από κλάσεις
  - Σαν γενεαλογικό δέντρο κλάσεων από πιο γενικές προς πιο ειδικές κλάσεις.

- Στη Java όλες οι κλάσεις ανήκουν στην ίδια ιεραρχία.
  - Στην κορυφή της ιεραρχίας είναι η κλάση Object.

Η βασική κλάση

```
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee( ) { … }

    public Employee(String theName, Date theDate) { … }

    public Employee(Employee originalObject) { … }

    public String getName( ) { … }
    public void setName(String newName) { … }

    public Date getHireDate( ) { … }
    public void setHireDate(Date newDate) { … }

    public String toString() { … }

}
```

Η παράγωγη κλάση HourlyEmployee

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { … }

    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours) { … }

    public HourlyEmployee(HourlyEmployee originalObject) { … }

    public double getRate( ) { … }
    public void setRate(double newWageRate) { … }

    public double getHours( ) { … }
    public void setHours(double hoursWorked) { … }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){ … }
}
```

Νέα πεδία για την HourlyEmployee

Μέθοδος getPay υπολογίζει το μηνιαίο μισθό

Η παράγωγη κλάση SalariedEmployee

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            Date theDate, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( ) { … }
}
```

Νέα πεδία για την SalariedEmployee

Μέθοδος getPay υπολογίζει το μηνιαίο μισθό. Διαφορετική από την προηγούμενη

```
public class Example1
{
    public static void main(String[] args)
    {
        HourlyEmployee alice = new HourlyEmployee("Alice",
            new Date("January", 1, 2004), 50.50, 160);

        SalariedEmployee bob = new SalariedEmployee("Bob",
            new Date("January", 1, 2005), 20000);

        System.out.println("Alice: "
                           + alice.getName() + " "
                           + alice.getHireDate() + " "
                           + alice.getPay());

        System.out.println("Bob: "
                           + bob.getName() + " "
                           + bob.getHireDate() + " "
                           + bob.getPay());

    }
}
```

Μέθοδοι της Employee

Μέθοδοι των παράγωγων κλάσεων

Constructor

```
public class Employee
{
    private String name = "default";
    private Date hireDate = new Date(11,4,2013);


    public Employee()
    {
       System.out.println("empty contructor");
    }

    public Employee(String theName, Date theDate)
    {
       name = theName;
       hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        super(theName, theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

Με τη λέξη κλειδί super αναφερόμαστε στην βασική κλάση.

Εδώ καλούμε τον constructor της Employee με ορίσματα το όνομα και την ημερομηνία.

Ο constructor super μπορεί να κληθεί μόνο στην αρχή της μεθόδου.

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

Τυπώνει “empty constructor”

Αν δεν υπάρχει κλήση του constructor καλείται αυτόματα ο constructor χωρίς ορίσματα.

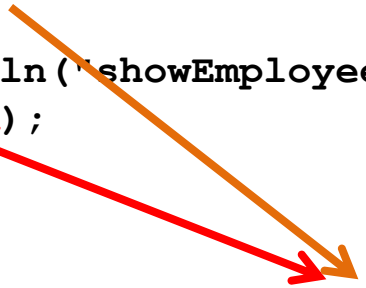
Αν δεν υπάρχει constructor χωρίς ορίσματα παίρνουμε λάθος στην εκτέλεση.

Πολλαπλοί τύποι

- Ένα αντικείμενο της παράγωγης κλάσης έχει και τον τύπο της βασικής κλάσης
  - Ένας HourlyEmployee είναι και Employee
  - Υπάρχει μία is-a σχέση μεταξύ των κλάσεων (Hourly Employee is a Employee).

- Αυτό μπορούμε να το εκμεταλλευτούμε χρησιμοποιώντας την βασική κλάση όταν θέλουμε να χρησιμοποιήσουμε κάποια από τις παράγωγες αλλά δεν ξέρουμε ποια εκ των προτέρων.

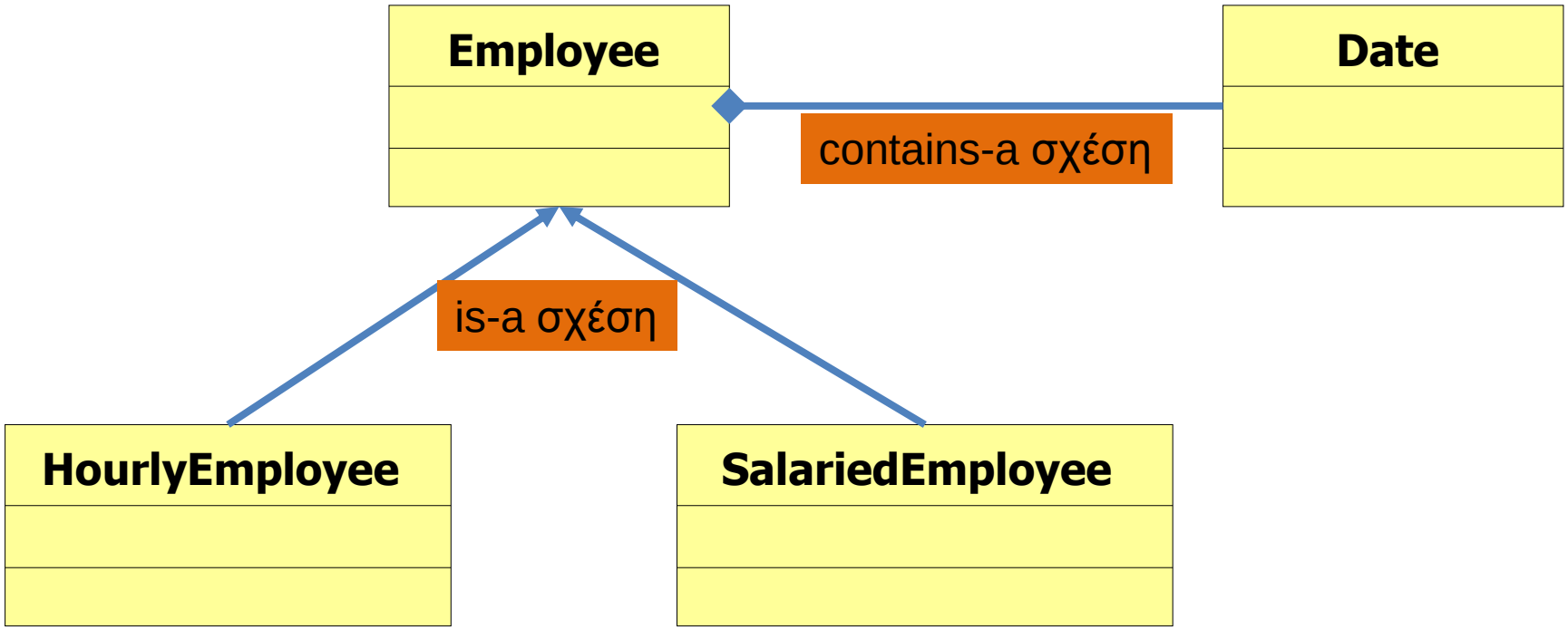
```
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                    new Date("January", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                               new Date("February", 1, 2003), 50.50, 40);

        System.out.println("joe's longer name is " + joe.getName( ));

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject.getName( ));
            System.out.println(employeeObject.getHireDate( ));
    }
}
```



Μπορούμε να καλέσουμε τη μέθοδο και με HourlyEmployee και με SalariedEmployee γιατί και οι δύο είναι και Employee.

UML διάγραμμα

- Αναπαράσταση κληρονομικότητας



Protected μέλη

- Οι παράγωγες κλάσεις έχουν πρόσβαση σε όλα τα public πεδία και μεθόδους της γενικής κλάσης.
- ΔΕΝ έχουν πρόσβαση στα private πεδία και μεθόδους.
  - Μόνο μέσω public μεθόδων set* και get*

- Protected: αν κάποια πεδία και μέθοδοι είναι protected μπορούν να τα δουν όλοι οι απόγονοι της κλάσης.
  - Το βιβλίο δεν το συνιστά.

- Package access: αν δεν προσδιορίσετε public, private, ή protected access τότε η default συμπεριφορά είναι ότι η μεταβλητή είναι προσβάσιμη από άλλες κλάσεις μέσα στο ίδιο πακέτο.

Employee

```
public class Employee
{
    private String name = "default";
    private Date hireDate = new Date(11,4,2013);


    public Employee()
    {
       System.out.println("empty contructor");
    }

    public Employee(String theName, Date theDate)
    {
       name = theName;
       hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        name = theName;
        hireDate = new Date(theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

Χτυπάει λάθος η πρόσβαση σε private πεδία.

Employee

```
public class Employee
{
    protected String name = "default";
    protected Date hireDate = new Date(11,4,2013);


    public Employee()
    {
       System.out.println("empty contructor");
    }

    public Employee(String theName, Date theDate)
    {
       name = theName;
       hireDate = new Date(theDate);
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month


    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        name = theName;
        Date = new (theDate);
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

ΟΚ η πρόσβαση σε protected πεδία.

Υπέρβαση μεθόδων (method overriding)

- Μία μέθοδος που ορίζεται στην βασική κλάση μπορούμε να την ξανα-ορίσουμε στην παράγωγη κλάση με διαφορετικό τρόπο
  - Παράδειγμα: η μέθοδος `toString()`. Την ξανα-ορίζουμε για κάθε παραγόμενη κλάση ώστε να τυπώνει αυτό που θέλουμε
  - Αυτό λέγετε υπέρβαση της μεθόδου (method overriding).
- Η υπέρβαση των μεθόδων είναι διαφορετική από την υπερφόρτωση.
  - Στην υπερφόρτωση αλλάζουμε την υπογραφή της μεθόδου.
  - Εδώ έχουμε την ίδια υπογραφή, απλά αλλάζει ο κώδικας της παράγωγης κλάσης.

```
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee( ) { … }

    public Employee(String theName, Date theDate) { … }

    public Employee(Employee originalObject) { … }

    public String getName( ) { … }
    public void setName(String newName) { … }

    public Date getHireDate( ) { … }
    public void setHireDate(Date newDate) { … }

    public String toString()
    {
        return (name + " " + hireDate.toString( ));
    }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee( ) { … }

    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours) { … }

    public HourlyEmployee(HourlyEmployee originalObject) { … }

    public double getRate( ) { … }
    public void setRate(double newWageRate) { … }

    public double getHours( ) { … }
    public void setHours(double hoursWorked) { … }

    public double getPay( ) {
        return wageRate*hours;
    }

    public String toString( ){
       return (getName( ) + " " + getHireDate( ).toString( )
            + "\n$" + wageRate + " per hour for " + hours + " hours");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            Date theDate, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( )
    {
        return (getName( ) + " " + getHireDate( ).toString( )
                                  + "\n$" + salary + " per year");
    }
}
```

```
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee( ) { … }

    public SalariedEmployee(String theName,
                            Date theDate, double theSalary) { … }

    public SalariedEmployee(SalariedEmployee originalObject ) { … }

    public double getSalary( ) { … }
    public void setSalary(double newSalary) { … }

    public double getPay( )
    {
        return salary/12;
    }

    public String toString( )
    {
        return (super.toString( ) + "\n$" + salary + " per year");
    }
}
```

Έτσι καλούμε την toString της βασικής κλάσης

Παράδειγμα

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
        HourlyEmployee han = new HourlyEmployee("Han",
                                   `new Date(1, 1, 2011), 50.50, 40);
        Employee eve = new Employee("Eve", new Date(1,1,2012));

        System.out.println(eve);

        System.out.println(sam);

        System.out.println(han);
       }
}
```

Καλεί τη μέθοδο της Employee

Καλεί τη μέθοδο της SalariedEmployee

Καλεί τη μέθοδο της HourlyEmployee

Υπέρβαση και αλλαγή επιστρεφόμενου τύπου

- Μια αλλαγή που μπορούμε να κάνουμε στην υπογραφή της κλάσης που υπερβαίνουμε είναι να αλλάξουμε τον επιστρεφόμενο τύπο σε αυτόν μιας παράγωγης κλάσης
  - Ουσιαστικά δεν είναι αλλαγή αφού η παράγωγη κλάση έχει και τον τύπο της γονικής κλάσης.

```
public class Employee
{
        private String name;
        private Date hireDate;

        public Employee createCopy()
        {
          return new Employee(this);
        }
}
```

```
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee createCopy()
    {
        return new HourlyEmployee(this);
    }
}
```

Ο επιστρεφόμενος τύπος αλλάζει από Employee σε HourlyEmployee στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο HourlyEmployee είναι και Employee

```
public class SalariedEmployee extends
Employee
{
    private double salary; //annual

    public SalariedEmployee createCopy()
    {
        return new SalariedEmployee(this);
    }

}
```

Ο επιστρεφόμενος τύπος αλλάζει από Employee σε SalariedEmployee στην υπέρβαση. Ουσιαστικά όμως δεν υπάρχει αλλαγή μιας και κάθε αντικείμενο SalariedEmployee είναι και Employee

toString και equals

- Είπαμε ότι η Java για κάθε αντικείμενο «περιμένει» να δει τις μεθόδους toString και equals
  - Αυτό σημαίνει ότι οι μέθοδοι αυτές ορίζονται στην κλάση Object που είναι ο πρόγονος όλων το κλάσεων και κάθε νέα κλάση μπορεί να τις υπερβεί (override).
  - Είδαμε παραδείγματα πως υπερβήκαμε την μέθοδο toString.

equals

- Η equals στην κλάση Object ορίζεται ως:
  - `public boolean equals(Object other)`
- Για την κλάση Employee θα την ορίσουμε ως:
  - `public boolean equals(Employee other)`
- Αλλάζουμε την υπογραφή της κλάσης, άρα δεν κάνουμε υπέρβαση, αλλά υπερφόρτωση της equals
  - Πως θα την ορίσουμε ώστε να κάνουμε υπέρβαση?

Overriding equals

```
public class Employee
{
    private String name;
    private Date hireDate;

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass( ) != otherObject.getClass( ))
            return false;
        else
        {
            Employee otherEmployee = (Employee)otherObject;
            return (name.equals(otherEmployee.name)
               && hireDate.equals(otherEmployee.hireDate));
        }
    }
}
```

getClass: μέθοδος της Object, επιστρέφει μια αναπαράσταση της κλάσης του αντικειμένου

Downcasting: μετατροπή ενός αντικειμένου από μια υψηλότερη σε μία χαμηλότερη κλάση

Το downcasting δεν είναι πάντα δυνατόν και αν δεν γίνει σωστά μπορεί να προκαλέσει λάθη κατά την εκτέλεση του προγράμματος

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                SalariedEmployee eve2 = (SalariedEmployee)eve;
                if (sam.getHireDate().equals(eve2.getHireDate())){
                        System.out.println("Same hire date");
                }else{
                        System.out.println("Different hire date");
                }
        }
}
```

Στην περίπτωση αυτή θα μας χτυπήσει λάθος στο τρέξιμο παρότι χρησιμοποιούμε μόνο την κοινή μέθοδο getHireDate(). Το πρόγραμμα προβλέπει ότι μπορεί να υπάρχει πρόβλημα.

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                method(sam,sam);

        }

        private static void method(SalariedEmployee sEmp, Employee emp){
                SalariedEmployee sEmp2 = (SalariedEmployee) emp;

                if (sEmp.getHireDate().equals(sEmp2.getSalary())){
                        System.out.println("Same Salary");
                }else{
                        System.out.println("Different salary");
                }
        }
}
```

Στην περίπτωση αυτή το downcasting δεν χτυπάει λάθος γιατί μπορεί να καλέσουμε σωστά την μέθοδο με SalariedEmployee αντικείμενο

Downcasting

```
public class DowncastingExample
{
    public static void main(String[] args)
    {
                SalariedEmployee sam = new SalariedEmployee("Sam",
                                    new Date(1, 1, 2010), 100000);
                Employee eve = new Employee("Eve", new Date(1,1,2012));

                method(sam,eve);

        }

        private static void method(SalariedEmployee sEmp, Employee emp){
                SalariedEmployee sEmp2 = (SalariedEmployee) emp;

                if (sEmp.getHireDate().equals(sEmp2.getSalary())){
                        System.out.println("Same Salary");
                }else{
                        System.out.println("Different salary");
                }
        }
}
```

Αν όμως την καλέσουμε με αντικείμενο Employee θα πάρουμε λάθος

```
import java.util.Random;

public class DowncastingExample2
{
    public static void main(String[] args)
    {
        SalariedEmployee[] sEmployees = new SalariedEmployee[4];
        sEmployees[0] = new SalariedEmployee("employee 100",new Date(1,1,2015),1000);
        sEmployees[1] = new SalariedEmployee("employee 101",new Date(2,1,2015),2000);
        sEmployees[2] = new SalariedEmployee("employee 102",new Date(3,1,2015),3000);
        sEmployees[3] = new SalariedEmployee("employee 103",new Date(4,1,2015),4000);

        SalariedEmployee rand = (SalariedEmployee)randomSelection(sEmployees);
        System.out.println(rand);
        System.out.println("Salary per month " + rand.getPay());
    }

    private static Employee randomSelection(Employee[] employees){
        Random rndGen = new Random();
        int r = rndGen.nextInt(employees.length);
        return employees[r];
    }
}
```

Σε τι μας χρειάζεται το downcasting?

Θέλουμε να καλέσουμε την μέθοδο getPay για τυπώσουμε τον μηνιαίο μισθό. Χρειαζόμαστε downcasting

Έχουμε μια γενική μέθοδο randomSelection που επιλέγει ένα τυχαίο στοιχείο από ένα πίνακα με Employee. Θέλουμε να την χρησιμοποιήσουμε σε ένα πίνακα με SalariedEmployee

Upcasting

- Η ανάθεση στην αντίθετη κατεύθυνση (upcasting) μπορεί να γίνει χωρίς να χρειάζεται casting
  - Μπορούμε να κάνουμε μια ανάθεση x = y δύο αντικειμένων αν:
    - τα δύο αντικείμενα να είναι της ίδιας κλάσης ή
    - η κλάση του αντικειμένου που ανατίθεται (y) είναι απόγονος της κλάσης του αντικειμένου στο οποίο γίνεται η ανάθεση (x)

- Για παράδειγμα, ο παρακάτω κώδικας δουλεύει χωρίς πρόβλημα:
  - `Employee anEmployee;`
  - `Hourly Employee hEmployee = new HourlyEmployee();`
  - `anEmployee = hEmployee;`

```
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                    new Date("January", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                                new Date("February", 1, 2003), 50.50, 40);

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject.getName( ));
            System.out.println(employeeObject.getHireDate( ));
    }
}
```

Όταν καλούμε την **showEmployee** έμμεσα κάνουμε τις αναθέσεις:

`employeeObject = joe`

`employeeObject = sam`

```
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                    new Date("January", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                                new Date("February", 1, 2003), 50.50, 40);

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject);
    }
}
```

Τι θα τυπώσει η **showEmployee** όταν την καλέσουμε με ορίσματα το joe και το sam?
Ποια μέθοδος **toString** θα κληθεί?

```
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                    new Date("January", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                                new Date("February", 1, 2003), 50.50, 40);

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject)
    {
            System.out.println(employeeObject);
    }
}
```

Θα καλέσει την **toString** της κλάσης του αντικειμένου που περνάμε σαν όρισμα (**HourlyEmployee** ή **SalariedEmployee**) και όχι την κλάση που εμφανίζεται στον ορισμό της παραμέτρου (**Employee**).

Ο μηχανισμός αυτός ονομάζεται late binding (και/ή πολυμορφισμός)

Late Binding (καθυστερημένη δέσμευση)

- Στη Java ο κώδικας που θα εκτελεστεί όταν καλούμε μια μέθοδο δεν καθορίζεται (δεσμεύεται) όταν γίνεται η μεταγλώττιση του προγράμματος (early binding) αλλά όταν γίνει η κλήση της μεθόδου από το αντικείμενο (late binding - καθυστερημένη δέσμευση)
  - Τη στιγμή εκείνη ξέρουμε ακριβώς την κλάση του αντικειμένου που καλεί την μέθοδο (π.χ., **`Employee`**, **`HourlyEmployee`** ή **`SalariedEmployee`**) και μπορούμε να εκτελέσουμε τον κατάλληλο κώδικα.
  - Το κάθε αντικείμενο έχει πληροφορία για τον ορισμό (κώδικα) των μεθόδων του.

- Ο μηχανισμός του late binding εφαρμόζεται για όλες τις μεθόδους στην Java (σε αντίθεση με άλλες γλώσσες προγραμματισμού).

Παράδειγμα

```
public class Example3
{
    public static void main(String[] args)
    {
        Employee employeeArray[] = new Employee[3];

        employeeArray[0] = new Employee("alice",
                                        new Date(1,1,2010));

        employeeArray[1] = new HourlyEmployee("bob",
                                  new Date(1,1,2011), 20, 160);

        employeeArray[2] = new SalariedEmployee("charlie",
                                  new Date(1,1,2012), 24000);

       for (int i = 0; i < 3; i ++){
                System.out.println(employeeArray[i]);
        }
    }
}
```

Για κάθε στοιχείο του πίνακα καλείται διαφορετική μέθοδος toString ανάλογα με το αντικείμενο που τοποθετήσαμε σε εκείνη τη θέση

```
public class mySale
{
    protected String name;
    protected double price;

    public mySale(String theName, double thePrice){
        name = theName;
        price = thePrice;
    }

    public String toString( ){
        return (name + " Price and total cost = $" + price);
    }

    public double bill( ){
        return price;
    }

    public boolean equalDeals(mySale otherSale){
        return (name.equals(otherSale.name)
                && this.bill( ) == otherSale.bill( ));
    }

    public boolean lessThan (mySale otherSale){
        return (this.bill( ) < otherSale.bill( ));
    }
}
```

Σύμφωνα με το βιβλίο δεν συνίσταται η χρήση της protected αλλά την χρησιμοποιούμε για απλότητα στο παράδειγμα

```
public class myDiscountSale extends mySale
{
    private double discount;

    public myDiscountSale(String theName,
                                  double thePrice, double theDiscount)
    {
        super(theName, thePrice);
        discount = theDiscount;
    }

    public double bill( )
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }


    public String toString( )
    {
        return (name + " Price = $" + price
                + " Discount = " + discount + "%\n"
                + "   Total cost = $" + bill( ));
    }
}
```

Υπέρβαση της μεθόδου **bill()**

Δεν έχουμε υπέρβαση των μεθόδων **equalDeals** και **lessThan**

```
public class myLateBindingDemo
{
    public static void main(String[] args)
    {
        mySale simple = new mySale("floor mat", 10.00);//One item at $10.00.
        myDiscountSale discount = new myDiscountSale("floor mat", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(simple);
        System.out.println(discount);

        if (discount.lessThan(simple))
            System.out.println("Discounted item is cheaper.");
        else
            System.out.println("Discounted item is not cheaper.");

        mySale regularPrice = new mySale("cup holder", 9.90);//One item at $9.90.
        myDiscountSale specialPrice = new myDiscountSale("cup holder", 11.00, 10);
                                  //One item at $11.00 with a 10% discount.

        System.out.println(regularPrice);
        System.out.println(specialPrice);

        if (specialPrice.equalDeals(regularPrice))
            System.out.println("Deals are equal.");
        else
            System.out.println("Deals are not equal.");
    }
}
```

Οι **lessThan** και **equalDeals** κληρονομούνται από την **mySale**

Με το μηχανισμό του late binding στην κλήση τους ξέρουμε ότι το αντικείμενο που τις καλεί είναι τύπου **myDiscountSale**

Ξέρουμε λοιπόν ότι όταν εκτελούμε τον κώδικα της **lessThan** και **equalDeals** η μέθοδος **bill()** που θα πρέπει να καλέσουμε είναι αυτή της **myDiscountSale** ενώ για το **otherSale.bill()** είναι αυτή της **mySale**